

cxterm: A Chinese Terminal Emulator for the X Window System

MAN-CHI PONG

Department of Computer Science, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong

AND

YONGGUANG ZHANG

Department of Computer Sciences, Purdue University, West Lafayette, IN 47907, U.S.A.

SUMMARY

Computing involving data in a logo-syllabic oriental language like Chinese is more difficult than in English. Many attempts have been made to develop bilingual or multilingual processing systems.¹⁻³ Most of them are based on microcomputers like the IBM-PC. With the widespread use of workstations and window systems, much better multi-lingual processing environments can be provided.

This paper describes the design and implementation of the program *cxterm*, a Chinese terminal emulator for the X Window System. We discuss the representation of multi-byte international characters, the problems of Chinese character input and output, the pros and cons of various approaches, and the design decisions for *cxterm*. A feature of *cxterm* is its independence of input methods. A user can incorporate new input methods into *cxterm* at run-time, without changing the program code. We also compare our approach with related work in multi-lingual input/output in X, and describe how *cxterm* performs better in terms of efficiency, flexibility, and user-friendliness.

KEY WORDS X window system Terminal emulator Multilingual computing Chinese input and output Input method User interface

INTRODUCTION

The *X window system* (X for short) has become the *de facto* standard window system in workstations. It operates on a server-client model. The X server program running in a workstation manages its input and output devices: screen, keyboard, and mouse. Other X client programs, including various window managers, accept user input and request output services from the X server through different interprocess communication channels using the X protocol.⁴ The services include creating, deleting, repositioning and resizing windows; directing keyboard input to a particular window; displaying characters and drawing graphics in windows; etc. The display of a character is actually done by drawing the bitmap image of the character in the window. Collections of characters for displaying text in X are known as *fonts*. A font typically contains character images that have the same style of appearance and look nice when displayed or printed together. X supports the use of different fonts. By changing the font, the same string of characters can be displayed in different ways.

Received 24 June 1991
Revised 5 May 1992

0038-0644/92/100809-18\$14.00
© 1992 by John Wiley & Sons, Ltd.

X supports what is called the *graphical user interface* (GUI): the user sees a graphical output and the user can input data via a keyboard and a pointing device such as a mouse. In order to run the numerous existing utilities and applications based on the *character user interface* (CUI) in X, an X client program called *xterm* was released with X version 10, the first public version of X. The program *xterm* emulates an ASCII character terminal in X. It behaves like a hard-wired character-based terminal, except that resizing the *xterm* window is allowed and the column and/or row numbers of the emulated terminal will change accordingly. However, like most hard-wired terminals, *xterm* cannot cope with multi-byte international character sets.

In this paper, we describe *exterm*, a Chinese terminal emulation program for X, which enables the display and input of Chinese characters. For brevity, we shall use the Chinese 'pronunciation spelling' *hanzi* to stand for *Chinese character(s)*. Note that *hanzi* is used as both a singular and a plural noun.

In the following sections, we describe the problem of computing involving multi-byte international characters. In particular, we concentrate on the treatment of *hanzi*. We describe the difficulties encountered and the important design decisions in developing *exterm*. A significant feature of *exterm* is that different *hanzi* input methods are supported; the addition of a new input method does not require any change in the *exterm* program. Finally, we discuss related works in terminal emulation for multi-byte character computing, and the extension of *exterm* to input Chinese phrases.

MULTI-BYTE CHARACTER REPRESENTATIONS

Computers traditionally process characters represented in single bytes only, such as EBCDIC or ASCII. Nowadays eight-bit European language character sets are commonly supported by vendors. For the Chinese, Japanese or Korean languages, the character sets contain thousands of logo-syllabic characters that are mainly originated from Chinese characters. A *code* of more than one byte is used to represent each character. The corresponding national standard bodies have defined standard character sets that contain the commonly used characters of the respective languages.

For the Chinese language, the most widely used official standard character set in Mainland China is GB2312-80 (GB for short).⁵ It contains 6763 *simplified hanzi*,* each represented by a two-byte code of the form 0xxxxxx 0xxxxxx (where *x* can be either 0 or 1). In Taiwan and Hong Kong, most commercial programs use the so-called *Big-5* character set.⁶ It contains over 13,000 *traditional hanzi*, each represented by a two-byte code of the form 0xxxxxx xxxxxxx.

A common way to represent more than one character set in a stream of bytes is to use an *escape character sequence* to signify the start of another character set, and sometimes, also to signify the end of a character set. This is the scheme commonly used in Japanese language processing software,⁷ where the Japanese characters are represented in ASCII, kana and kanji† character sets.

* Currently, *simplified hanzi* is used daily in Mainland China while *traditional hanzi* is used in Taiwan and Hong Kong. However, even in Mainland China, knowledge of traditional *hanzi* is still required to study pre-1957 writings. (The first standard on written simplified *hanzi* was introduced in 1957.)

† *Kanji* is the collective term for Chinese characters in the Japanese language. The number of *kanji* in Japanese is much less than the number of *hanzi* in Chinese.

For hanzi processing, another scheme is often used because we only deal with one seven-bit character set (ASCII) and one two-byte hanzi character set (GB or Big-5). We can simply set the most significant bit (the *high-bit*) of the first byte of the hanzi code to 1 (or *on*). We call this the *high-bit-on* scheme. A program, on processing a stream of bytes, can differentiate the characters by examining the high-bit of a byte. If the high-bit is on, then this byte and the following byte together represent a hanzi; otherwise, this byte is simply a seven-bit ASCII character. In Mainland China, it is common to set the high-bits of both bytes of a GB character code to 'on'. In contrast, Big-5 hanzi code uses all eight bits of the second byte, and thus the second byte may have its high-bit 'on' or 'off'. Since the high-bit of the first byte is sufficient to differentiate a hanzi code from a 7-bit ASCII code, it does not matter whether the high-bit of the second byte is on or off.

Using the high-bit-on scheme, no escape character sequences are needed. This minimizes the size of the byte stream representing mixed ASCII and Chinese characters. There is another advantage of not using escape character sequences. When the character image width of the Chinese font is twice that of the ASCII font, the mixed characters displayed along a line have one-to-one correspondence with the bytes in the byte stream. For example, if a buffer is used to store the characters displayed in a window, and a character position in the window is given, the one- or two-byte code corresponding to the character can be directly located in the buffer. (As a side note, to display mixed hanzi and ASCII characters nicely along lines, the character image heights of both fonts should be the same.)

The biggest problem of the high-bit-on scheme is that many old programs use the high-bit-on status to represent another add-on property of the seven-bit ASCII characters, e.g. to display the characters in highlight mode. Some programs simply strip the eighth bit, e.g. some electronic mail transfer programs. To make the high-bit-on scheme work in these programs, we can modify them to become *eight-bit-clean*, i.e. the high-bit of a byte of a character code is not used for other purposes. Fortunately, many programs from vendors have become eight-bit-clean in order to process eight-bit codes of European languages. Moreover, eight-bit binary file encoding/decoding programs, such as uuencode/uudecode, are available in almost all operating systems. To send a Chinese text file using communication programs that strip off the eighth bit, one can encode the file into an ASCII file before sending, and decode back to the original high-bit-on file upon receiving. In short, the obstacles to the high-bit-on scheme are disappearing.

HANZI INPUT AND OUTPUT PROBLEMS

Since the number of hanzi is so large, special *input methods* are required to input hanzi using the standard QWERTY keyboard. Each such input method has an associated *input table*, which is often called a *dictionary*. An input table entry consists of one or more hanzi represented by a sequence of keystroke symbols. There may be more than one hanzi in each entry because a keystroke sequence usually cannot uniquely identify individual hanzi.

To input hanzi, the user types a sequence of keystrokes, and an input processing module captures the keystrokes in a buffer and matches the sequence against the input table. A valid keystroke sequence eventually matches an entry in the input

table. The input processing module then displays the hanzi in the entry as a *choice-list* of candidate hanzi for the user to choose among.

Commonly used input methods are based on the pronunciation (or *pinyin*) or the radical structure (the shape) of hanzi, or both. Although there are many different input methods, none of them has gained universal acceptance, and many more are being developed.^{8,9} A Chinese input processing software module should therefore be adaptable to support different input methods, since different people may favour different input methods and since new input methods may be proposed.

Chinese is an ideographic language. For hanzi output, we need a way to draw the character image of any hanzi code on the screen. The widespread use of workstations with bitmap display hardware and graphics software solves this problem. Bitmaps of hanzi character images can be put into a font file, and today's workstations already support multiple character fonts. As the X window system becomes more and more popular, it is desirable to have Chinese processing software in X.¹⁰

Our first attempt* to add Chinese processing capability to X and to develop *exterm* was based on X10 (X version 10). However, X10 only supposed eight-bit character fonts. A brute force approach is to let the first byte of a two-byte code identify a font and the second byte identify the character image within that font. This would require a large number of fonts, and consequently the performance of X would seriously degrade when there is a lot of switching between fonts. Instead, we decided to enhance the X protocol⁴ by adding special X library routines and corresponding program code in the X server to open font files of two-byte hanzi code and to retrieve the character image directly. The use of such files did not follow the X10 font standard as there was no X10 font standard for 16-bit characters.

Soon after our effort on the display of hanzi in X10 was completed, X version 11 was announced. X11 supports 16-bit character fonts. This simplifies the task of *exterm* to display a hanzi. We can simply use the two-byte code of a hanzi as an index to retrieve its image in any 16-bit hanzi font.

APPROACHES TO HANZI INPUT PROCESSING IN X

X11 makes hanzi output as easy to produce as ASCII output. One remaining concern for hanzi processing in X is handling hanzi input in application programs.

Hanzi input processing in X can be done in the X server, in the X client program level, or by a special *input server* program. An input server is a special X client program running as a separate process. It implements some input methods, and other X client applications can communicate with it by sending input requests and receiving converted hanzi characters.¹¹

Adding input processing to the X server is undesirable because running Chinese processing software should not require a special extended version of the X server. On the other hand, input processing can be implemented in the X client program level in a simple way. The hanzi input conversion can be made part of the X client process, and the hanzi input processing area can be made part of the application window.

The input server approach has its advantage. By separating the input conversion processing from the application program, it requires only one input server process

* When the authors were with the Institute of Software, Academia Sinica, Beijing, China.

for multiple applications. However, this approach causes more inter-process communication overhead because the application program has to send hanzi input request to the input server, and the input server has to send the hanzi code back to the application. Moreover, it is harder to design an input server with a user-friendly interface. For example, it would often be necessary for a user to switch his attention between the application window and the input server window because hanzi input requires user interaction in the input server window when choosing the desired hanzi from a choice-list of candidates. It is unacceptable if the user has to move the mouse cursor back and forth between two windows. A user-friendly input server should be carefully designed to avoid this. Also, having many application programs sharing *one* input server raises the problem of concurrent access to the input server window. Since each application has its own independent input context in the server, the input server has to guarantee that hanzi input for several applications is routed properly. In summary, to avoid the complexity of the input server approach, we choose to do hanzi input processing in the X client program level.

We modify the X client program *xterm*, an ASCII terminal emulator, to become *cxterm*, a Chinese terminal emulator, and let it take care of hanzi input/output. The conversion of a keystroke sequence to the corresponding hanzi code is done in *cxterm* and is hidden from the application program running in *cxterm*. This approach provides a user-friendly character-based human-computer interface for many application programs. Existing character terminal based programs operating only on ASCII characters can run in *cxterm* without modifications. These programs can be modified to handle mixed ASCII and Chinese characters. For traditional screen-oriented programs such as editors, the modification to run in *cxterm* is simplified by the high-bit-on hanzi encoding scheme, because there is a one-to-one correspondence between the window buffer of character codes and the appearance of the characters in the window. The programs can simply output the byte stream of mixed characters, as before, to *cxterm*, which distinguishes hanzi from ASCII characters and displays them using different fonts appropriately.

We have also considered building a hanzi input library, comparable to the standard ASCII input library, so that any X client application program can be linked with the library. The library contains routines to convert an input keystroke sequence to hanzi code. The application calls the library routines during run-time whenever hanzi input is required. However, the library approach is more suitable for GUI-based applications. We would like to exploit the currently CUI-based application programs, e.g. an editor, to do hanzi processing. Using the library approach, we would have to modify the existing CUI-based programs to call different library routines to input and output hanzi characters wherever ASCII characters are handled. This would be tedious when compared to the *cxterm* approach. We therefore decided to build *cxterm* instead of building a hanzi input/output library.

REQUIREMENTS

The requirements we posed on *cxterm* are based on the consideration discussed above and the support provided by X11. Firstly, *cxterm* should be independent of any hanzi representation code. We plan to support only two-byte character codes so that we use the high-bit-on scheme, as opposed to an escape character sequence, to distinguish between ASCII and Chinese characters in a data stream. The Chinese

character set could be GB, Big-5, or any other with a two-byte code of the form 0xxxxxx xxxxxxx, where x is 0 or 1.

Secondly, cterm should provide efficient and flexible hanzi input handling. It should support multiple hanzi input methods, allow the user to switch among them, and allow him to define and add *any* input method without modifying the cterm program.

Moreover, cterm should support all the functionalities of the ASCII terminal emulator xterm, namely: repainting following exposure, resizing and scrolling of the cterm window, and cutting and pasting text among many cterm windows. It should also allow the use of different hanzi fonts.

Finally, cterm should be self-contained. That is, it should not depend on any extension of the X window systems, such as the internationalization extension in X11 Release 5.

USER INTERFACE

Cterm now runs in many brands of workstations running X11 Release 4 and Release 5. Figure 1 shows the appearance of the cterm window when cterm is invoked. It is similar to xterm, except that the bottom part of the window has been marked off by a horizontal line and serves as the hanzi input processing area (*input area* for short). The input area is not a sub-window. The user can input hanzi when the mouse pointer is *anywhere* inside the cterm window. The cterm program knows that only the portion of the window above the horizontal line is the emulated terminal area.



Figure 1. cterm window

Choosing hanzi input methods

When a cterm window is created, it is in the ASCII *input mode*, as indicated by the input area prompt message as seen in Figure 1. In this mode, cterm interprets keystrokes exactly as does xterm. The corresponding ASCII characters are entered into the input stream of the emulated terminal.* The application program currently running in cterm receives the input characters. The program could be a shell, the command interpreter in Unix, an editor, etc. Figure 2(a) shows the keystroke data flow in the ASCII input mode.

To input hanzi, the user chooses one of the input methods and cterm enters the corresponding hanzi input mode. The user's keystrokes are captured in an input buffer and cterm matches the captured keystroke sequence against the input table of the current input method. Figure 2(b) shows the keystroke data flow in hanzi input mode.

In the ASCII or any hanzi input mode, the user can press user-definable control keys to switch to other input methods. These control keys are defined and interpreted using X window system conventions.¹² Table I shows the default bindings of the control keys for switching to commonly used input methods.

The user can bind the switching keys in different ways for these input methods, or define new switching keys for new input methods. We use the X11 event translation mechanism to implement the key binding. The details can be found in X manuals.¹² Figure 3 shows the appearance of the cterm window when the input mode is changed to the *pinyin* input method. The input area prompt message indicates that cterm is currently in the pinyin input mode.

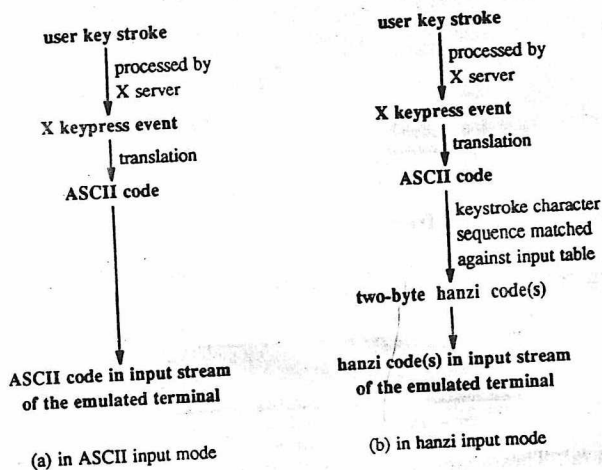


Figure 2. Data flow of the user keystroke

* Technically, it is the input stream of the pseudo-terminal-input/output-driver of the emulated terminal. We simply call it the *input stream*.

Table I.

Switching control key	Input method
F1	ASCII
F2	internal representation code
F3	pinyin + tone
F4	pinyin
F5	double-byte ASCII symbol
F6	double-byte punctuation symbol
F7	first- and last-stroke encoding
F8	telegraph code
F9	Cantonese pinyin

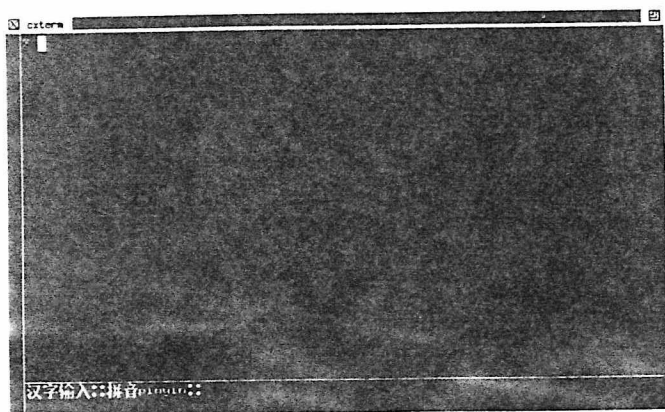


Figure 3. cxterm window in pinyin input mode

Choosing the desired hanzi from the choice-list

Figure 4 shows the input area immediately after the user presses 'a' as the first keystroke in pinyin input mode. There are many hanzi with a pinyin sequence

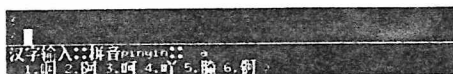


Figure 4. Input area of cxterm when 'a' is pressed

starting with 'a'. These hanzi form a *choice-list* ready to be displayed for the user to choose the desired hanzi. The maximum number of choices displayed in the input area, say n , is limited by the cxterm window width. We have tried the default value of $n = 7$ because studies had shown that people can handle about seven chunks of

data at a time.¹³ However, we found that the display of seven choices spans too widely, and most users cannot see them in one glance. Thus we now set the default $n = 6$. The user can override this by specifying the desired maximum number in the *input table definition file* associated with the input method (described in the next section).

If there are more choices before and/or after the currently displayed choices, the characters '<' and/or '>' are displayed *before the first* and/or *after the last* displayed choice, respectively. Figure 4 shows the *very first* six choices, thus only '>' appears at the end. To see the next n choices in the list, the user presses the keystroke '>' (or '.'—usually '.' is the non-shift state of the same key as '>'). To see the previous n choices in the list, the user presses the keystroke '<' (or ','). When the tail (or the head) of the list is reached, if the user attempts to move forward (or backward) by pressing '>' (or '<'), cterm warns the user via audio beep. The keystrokes '>', '<', '.', and ',' are collectively called the *choice-list traversal keys*.

When n choices are displayed, the user can press a *selection key* to make a choice. Once this selection is made, cterm puts the two-byte code of the chosen hanzi into the input stream, clears the input area, and re-displays the input area prompt message (as in Figure 3). The default selection keys are the space-bar and the digit keys '1' to 'n'. By default, pressing the space-bar is equivalent to pressing the key '1'. It is much easier to type the space-bar, so this is a handy way to select the first choice, which is often the hanzi with the highest occurring frequency among the choices. Both the traversal keys and the selection keys mentioned above are re-definable by the user in the corresponding input table definition file.

If the user does not make a choice, but presses the key 'i', then the pinyin sequence becomes 'ai' and the number of hanzi choices is reduced. cterm shows the first n hanzi of the new choice-list corresponding to 'ai'. Figure 5 shows the resulting input area.



Figure 5. Input area of cterm after 'a', 'i' are pressed

Any keystrokes that are not in the acceptable alphabet of the input method, such as ASCII punctuation marks and control characters, are simply passed directly into the input stream. We call such characters *pass-through characters*. It is convenient to have such characters pass directly into the input stream, because when the user inputs a mix of hanzi and pass-through characters, he need not switch to the ASCII input mode to input the pass-through characters and then switch back to the hanzi input mode to input additional hanzi.

When the hanzi and/or ASCII codes are passed into the input stream, the application program running in the cterm window receives them and can process them appropriately. If the program wants to output a hanzi in the cterm window, it can simply write the two-byte hanzi code to the standard output stream. cterm receives the code, uses it to retrieve the corresponding character image from the current hanzi font, and displays the hanzi image at the appropriate position in the window.

TEXTUAL AND COMPILED INPUT TABLES

A user can specify a hanzi input method in an input table definition file, say with file name 'inputMethodName.TIT', where TIT stands for *Textual Input Table*. At run-time, exterm loads the input table of the current input method into memory. The user's keystroke sequences are then matched against this input table. Figure 6 gives the essential details of the 'pinyin.TIT' file for the pinyin input method. The following paragraphs explain the format of the TIT file.

The keyword 'ENCODE:' defines the character set being used, e.g. 'GB' or 'Big5'.

The keyword 'PROMPT:' introduces the input area prompt message. The message is displayed in the first line of the input area to remind the user what the current input method is.

The keyword 'SELECTKEY:' defines the *selection keys* that are used to choose one hanzi from a list of many alternatives. Note that the first token specified is '1040'. This means that in order to select the first choice, the user can press either the key '1' or the space-bar (the ASCII space character has octal value 040). The other specified selection keys are separated by spaces. If the 'SELECTKEY:' line is not specified, the result is the same as if 'SELECTKEY: 1040 2 3 4 5 6' were entered. The user can specify more or fewer selection keys; e.g. the line 'SELECTKEY: 1040 2 3 4 5 6 7 8 0' defines 10 selection keys.

The keywords 'MOVERIGHT:' and 'MOVELEFT:' define the *choice-list traversal keys* for the user to traverse the choice-list.

The set of ASCII characters that form *valid keystroke sequences* for a certain input method is called the *acceptable alphabet* of the input method. In the TIT file, the keyword 'VALIDINPUTKEY:' is used to define such an alphabet.

The following part of the TIT file is the *dictionary* section of the input method. Introduced by the keyword 'BEGINDICTIONARY', this section contains all valid key-stroke sequences; and for each sequence, there is a corresponding choice-list of hanzi.

Owing to the large character set, a choice-list normally consists of more than one

```

ENCODE: GB
PROMPT: 汉字输入::拼音pinyin::
SELECTKEY: 1\040 2 3 4 5 6
MOVERIGHT: .>
MOVELEFT: .<
VALIDINPUTKEY: :abcdefghijklmnopqrstuvwxyz

```

BEGIN D I C T I O N A R Y

a 啊阿呵吁跳啊喂
ai 挨埃咳咬坎痊痼癌疮毫冤霞霞艾碍爱隘媛媛暖咬咬
an 鞍氨安厂广谄庵按饕佬按按按暗岸胺岸端
ang 肮昂盎
ao 熬熬熬霜嗷嗷遨莪葵登整昼枵媯媯懊懊澳澳啻葵葵
ba 芭捌叭扒吧吧八疤巴芭靶拔破菠拔葵乾把把把坝霸罢爸

Figure 6. Part of "pinyin.TIT" file

hanzi. A good input method always tries to minimize the lengths of the choice-lists, i.e. to reduce the number of duplicated hanzi matching a certain keystroke sequence.

For run-time efficiency, the user must use the utility program `tit2cit` to convert a TIT file `'inputMethodName.TIT'` to a well-organized internal format, called *Compiled Input Table* (CIT), in the `'inputMethodName.CIT'` file. Cxterm reads the CIT file, not the TIT file, to load the actual input table into memory. In short, TIT is designed for the user to specify and maintain an input method while CIT is designed for cxterm to use. For completeness, another utility program `cit2tit` is developed to convert a CIT file to the corresponding TIT file.

IMPLEMENTATION

The program `cxterm` is derived from `xterm`. As discussed in the second section, `cxterm` uses first byte high-bit-on to identify a hanzi, and no escape character sequence is used for switching between character sets.

`xterm` requires a *normal ASCII font*; `cxterm` requires an extra *normal hanzi font*. `xterm` supports an optional *bold ASCII font*; `cxterm` also supports an optional *bold hanzi font*. These two optional fonts can be used independently of each other. A characteristic of hanzi is that the width of the character images in a font is constant. Thus, the normal and bold hanzi images should have the same width. For nice-looking display, the character image width of the hanzi fonts should be twice that of the ASCII fonts. This also implies that fixed width ASCII fonts should be used.

For input processing, whenever the user presses a key, X traps it and notifies `cxterm` as a keystroke event. In hanzi input mode, `cxterm` captures the keystroke character in a buffer, and matches the buffer against the current input table. When there is a match and the user types a selection key, it means that a hanzi is successfully input by the user. The two-byte code of the hanzi are obtained from the input table and entered into the input stream. The two-byte codes in the input table have the first bytes high-bit-on already.

The input table of a hanzi input method is stored in an external CIT file. It is only loaded at run-time on demand. When `cxterm` is first started up, there is no in-core CIT. When the user presses a control key to switch to a hanzi input mode, `cxterm` loads the corresponding CIT file and it becomes the current in-core CIT. Since the number of different input methods used in a `cxterm` session is typically not large, a CIT read into memory is never discarded. If the desired input method has been used previously, then its CIT is already in-core, and `cxterm` need not load it again; it is simply set to be the current CIT. After the current CIT is set for a new input method, the keystroke buffer is cleared. Then `cxterm` captures any new keystrokes in the buffer and matches the buffer against the new CIT.

Data structures

The data structure of the hanzi input table in `cxterm` is carefully designed to be run-time efficient. It is implemented in the form of a *trie*.¹⁴ The CIT file contains an array representation of the trie, so that `cxterm` can read in the whole trie directly and traverse it efficiently.

Figure 7 shows the portion of the trie corresponding to the `'pinyin.TIT'` file in Figure 6. Each valid keystroke sequence defined in the input table has a matching

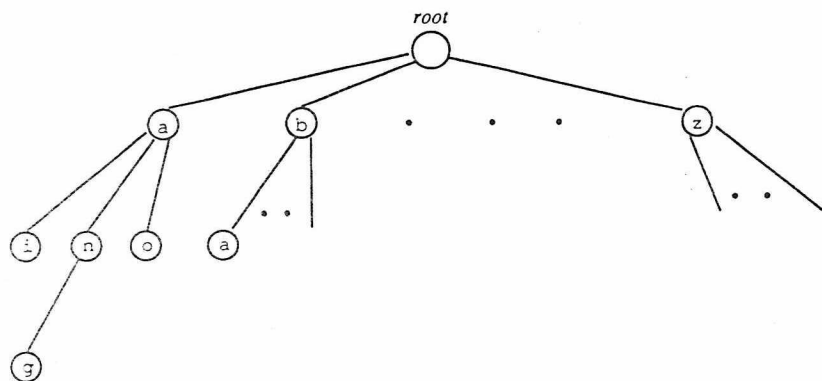


Figure 7. The trie structure of 'pinyin.TIT'

branch in the trie, with the first character in the sequence being the start node of the branch (which is one of the first level nodes below the root), and the last character in the sequence being the end node of the branch (which is either a leaf or an internal node in the trie). The end node of the branch contains a pointer to the choice-list of hanzi corresponding to the valid keystroke sequence.

Matching valid keystroke sequences against the input table

The input processing of *exterm* is independent of the input methods. New input methods can be independently incorporated, without modifying the program code of *exterm*.

This is achieved by employing a generic trie traversal routine. No matter from which CIT file the in-core trie is read and built, the trie has the same structure. Therefore *exterm* simply maintains a pointer to the *current* node of the in-core trie. At any moment during hanzi input mode, the keystroke character buffer is in one of two states: *start* state, when the buffer is empty, or *matched* state, when the buffer matches either a valid keystroke sequence or a prefix substring of some valid keystroke sequences. Correspondingly, when the buffer is in the *start* state, the trie traversal pointer points to the root, and when the buffer is in the *matched* state, it points to a node other than the root.

A design decision we made was to allow the user to choose a hanzi after any number of valid keystrokes are entered. In the *matched* state, the pointer points to a non-root node. Conceptually, *exterm* has to merge all the choice-lists of this node and its descendants to form a *merged choice-list*. Actually, by using the data structures shown in Figure 8, no such merging is needed. We store all the hanzi choice-lists in an array, and each trie node has two pointers pointing to the start and end of the merged choice-list of the node. In this way, hanzi input based on partial input of a valid keystroke sequence is possible.

As an example, for the pinyin input method, an input keystroke 'a' in the keystroke buffer causes merging of the choice-lists of valid keystroke sequences 'a', 'ai', 'an',

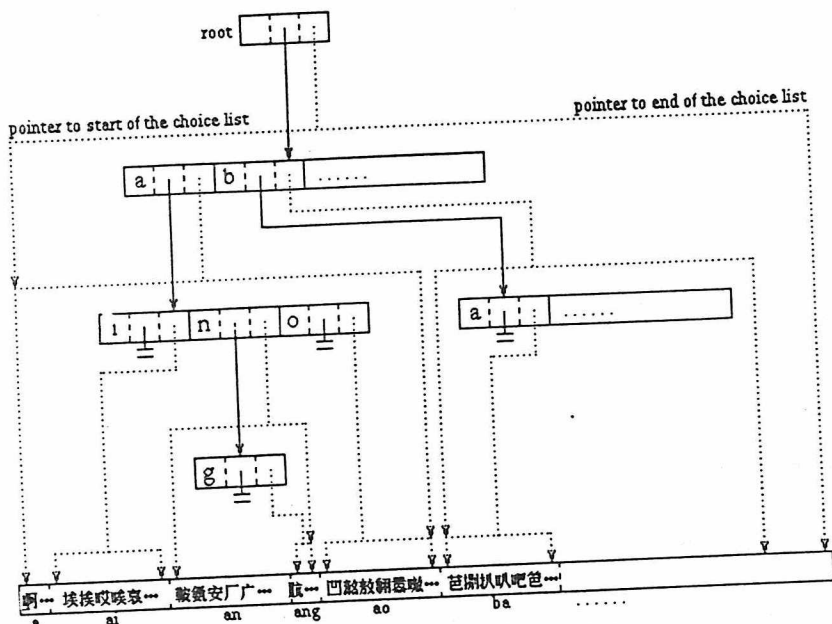


Figure 8. The implementation of trie structure

'ang', and 'ao' to give the current merged choice-list. The user can traverse this list and choose a hanzi out of the displayed choices. When a choice is made, *extern* puts the hanzi code into the input stream, clears the keystroke buffer, and returns to the start state.

On the other hand, if the user does not choose a hanzi when a merged choice-list is displayed, but instead presses a new keystroke that corresponds to a child of the current node, then the trie traversal pointer moves to that node. The merged choice-list and the input area are updated accordingly. For example, if the user enters 'i' after 'a', then the merged choice-list becomes that of 'ai' only. If the user enters 'n' after 'a', then the choice-list is formed from merging those of 'an' and 'ang'.

If the user enters any other keystroke in the acceptable alphabet, but the trie cannot be traversed downward, then *extern* echos a warning beep and ignores the keystroke. This is because the new keystroke cannot be concatenated to the keystroke buffer to form a prefix substring of any valid keystroke sequence. In such cases, the trie traversal pointer remains pointing at the current node.

The output processing of hanzi is straightforward with X11 8-bit and 16-bit fonts. *extern* calls the X library functions that use one-byte ASCII codes and two-byte hanzi codes as indices to obtain the character images from the ASCII and hanzi fonts, respectively, and displays them in the *extern* window.

Miscellaneous processing

As in *xterm*, *cxterm* uses a window buffer to store the characters shown in the window. The user can use the mouse to copy characters from the window. Actually, the characters are obtained from the window buffer. The user can use the mouse to paste the copied characters to the same or another window. This is performed by putting the character codes directly into the input stream of the target window.

In copying characters from a window, *cxterm* has to ensure that the begin mark of a selected text always falls on the first byte of a hanzi. If the mouse happens to point to the second byte of a hanzi, *cxterm* moves the mark to the first byte to ensure that the whole hanzi is covered. The same idea of getting only whole hanzi applies to the end mark of a selected region.

Refreshing part of a window can also be a problem if a border falls in the middle of a hanzi. Wrapping a long line around into two lines in a window also has to make sure that the line is not split in the middle of a hanzi. In short, *cxterm* has to check for the case of hanzi in all *xterm* actions and handle them properly.

SOME USEFUL PROGRAMS RUNNING IN CXTERM

Cxterm provides a satisfactory solution to the hanzi input/output problem of CUI-based programs. When we first released *cxterm*, we also released a screen-oriented editor *celvis* (or called *cvi*) for editing of text composed of mixed hanzi and ASCII characters. The editor *celvis* is derived from *elvis*, which is a public domain program having the same functionalities and interface of the standard editor *vi* in Unix systems.

The major modification for *celvis* is to ensure that when the editor cursor is on a hanzi, the cursor has to cover the whole hanzi, instead of just half of it. Meanwhile, the current byte pointer in the editing buffer corresponding to the cursor always points to the first byte of a hanzi. Moving the cursor left or right over a hanzi means moving the pointer in the buffer to the target ASCII byte or the first byte of the target hanzi. The editor *celvis* now supports almost all the functionalities of the editor *vi* for editing mixed hanzi and ASCII character text.

Later, we modified the public domain ASCII file viewing program 'less' for viewing mixed hanzi and ASCII characters. We call it *cless*. The modification was done quickly in three man-hours. This shows that with the support of *cxterm*, it is easy to convert existing CUI-based programs to handle hanzi.

COMPARISON WITH RELATED WORK

There are other efforts to make use of X to support multi-lingual information processing. *Mlx* (Multi-Lingual X)¹⁵ is an X window package designed to support multilingual characters, including hanzi. In this package, a terminal emulator called *mlxterm*, derived from *X11R2 xterm*, is used for display, and an input server program is used for input of multi-lingual characters.

Kterm is another X client program originally built to serve as a Japanese kanji terminal emulator.¹⁶ Kanji code input in *kterm* is performed through a separate input server program, *kinput*. Although the recent version of *kterm* supports the display of hanzi, hanzi input has not yet been incorporated into *kinput*.

Both *mlxterm* and *kterm* adopt the input server approach. In addition to the

communication overhead, the quality of the implementation of the input servers has large influence on the performance of the terminal emulators. For example, to input hanzi in an *mlxterm* window, the user must press some control key to set the *mlxterm* window to a certain input mode (called *MLR mode*), and move the mouse to the input server window. In the input server window, the user must select an input method first, and then type in the keystrokes for a hanzi according to the rules of the input method. It is cumbersome to do hanzi input if more than one *mlxterm* window is opened on the same screen, since input request by one *mlxterm* window automatically cancels the input mode of another *mlxterm*, and changing input method in one application implicitly forces all other applications to use the same input method. Also, it is not user-friendly because the user has to move the mouse back and forth, and turn some mode on and off in order to enter a hanzi.

The kanji conversion in *kinput* is better designed. When the conversion starts, *kinput* pops up a conversion window below the bottom side of the *kterm* window. To avoid mouse pointer movement, *kinput* also creates a transparent input-only window lying on top of the *kterm* window, and effectively grabs any input keystrokes in the *kterm* window. However, there is the overhead of having a window popped up each time whenever kanji input is required. In contrast, input processing in different *xterm* windows is independent of each other and is efficient.

Wnn is an input server that performs the conversion of a keystroke sequence to kanji, according to some pre-defined rules.¹⁷ There is also a Chinese version of *Wnn* called *cWnn*, which converts a keystroke sequence to hanzi. *Wnn/cWnn* defines a set of conversion protocol and interface routines that an application program must follow in order to perform any input. To use *Wnn/cWnn*, one has to modify an application program and link with a special library for *Wnn/cWnn*. As pointed out before, a sophisticated input server can be huge and complex. For the *Wnn/cWnn* input server, the size of the source code alone is about 1800 kilobytes. In comparison, the built-in hanzi input processing module of *xterm* is much simpler and more elegant; the size of the source code of the input module is only about 36 kilobytes.

There are internationalization (known as I18N) extensions of X libraries in *X11R5* that support input methods.¹⁸ However, different implementations of input methods may result in different I18N extensions of the X libraries. In contrast, *xterm* is only a client program and it requires no special I18N extension to the X libraries.

*Emacs*¹⁹ is a widely-used extensible editor and it can be run in an X window. There is a Japanese extension of *emacs*, called *nemacs*. The input processing in *nemacs* is through the *Wnn* input server. There are requests to add hanzi processing capabilities to *emacs*. However, *emacs* is a large piece of software of more than 10 megabytes of source code, and modifying it is not an easy task. Up till the time this paper is written, we have not heard of any successful incorporation of hanzi input/output processing *directly* into *emacs*. In contrast, a small patch of code of about five kilobytes for *emacs* has been developed. The code is mainly to make *emacs* eight-bit clean, and will be officially released with *emacs* version 19. The user can run this patched *emacs* in a *xterm* window and use it to edit hanzi.

CONCLUDING REMARKS

The *xterm* approach described in this paper overcomes the drawbacks of many input server approaches to Chinese input processing. The input processing in *xterm* is

efficient and user-friendly. The separation of the specification of input methods from input processing provides flexibility for the users to develop and use their favourite input methods. This is achieved by dynamic loading of compiled input tables.

Although our work is targeted at Chinese language processing, the approach employed should be useful in other bilingual processing systems. For example, the input methods for inputting Japanese or Korean characters can be specified in TIT files, as done for Chinese characters. Also, *extern* support the use of different font files. As long as two-byte internal codes are used to represent Japanese or Korean characters and to index into the corresponding font file to retrieve the character images, we do not need to modify *extern* for displaying Japanese or Korean characters.

We have released *extern* in 1991 as free software.* A future enhancement of *extern* should involve extending hanzi input processing to support input of multi-hanzi phrases. This would speed up the input process because inputting one hanzi at a time is too slow. Since many Chinese phrases are frequently used, one can define some valid keystroke sequences to represent these Chinese phrases. Then the input processing module can capture the user keystroke sequence in a buffer and search for the corresponding choice-list of hanzi phrases. The user then selects the desired phrase from the choice-list, and the input processing module puts the internal hanzi codes of the selected phrase into the input stream.

One simple way to incorporate hanzi phrase input without substantial change of *extern* is to allow the user to define the mapping of a keystroke sequence to a choice-list of phrases in the TIT file. Figure 9(a) shows a portion of such an input table. As for single hanzi, the same TIT file specification is used, except that there is one more section for specifying keystroke sequences and the corresponding phrases separated by commas. This section is introduced by the keyword 'BEGINPHRASES'. The CIT file corresponding to this TIT file contains the same trie data structure, except that the choice-lists contain a mix of single hanzi and hanzi phrases. The same trie traversal and hanzi selection mechanism can be used.

The user interface is also similar, as illustrated in Figure 9(b). The user has entered the keystroke sequence 'zhong'. The input area shows a choice-list of single hanzi and hanzi phrases. If the user presses the selection key '4', *extern* will put the internal codes of the chosen two-hanzi phrase into the input stream. This is more effective than input hanzi one by one. We are in the process of extending *extern* to support phrase input in this direction.

ACKNOWLEDGEMENT

Extern would not exist without the effort of all those who have contributed to X and *xterm*. We thank all the researchers and developers in MIT Project Athena and the X Consortium. We thank Stephen Kirkendall for developing the program *elvis*, and Mark Nudelman for the program *less*. We also thank Stephen Thebaut of the University of Florida, now visiting the Hong Kong University of Science and Technology, and the referees, who provided useful comments to improve the final version of the paper.

* *extern* is available from cs.purdue.edu via Internet anonymous ftp under directory pub/yzg/. There is a version for X11R4 and another version for X11R5. The X11R5 version is also part of the X11R5 contributed software distributed by M.I.T. X Consortium.

BEGINDICTIONARY

zhong 中钟终

...

BEGINPHRASES

zhong1 中国, 中国人, 中华, 中间, 中文, 中心, 中学

zhong2 钟点, 钟头

zhong3 终于, 终日, 终究

...

(a) Part of an input table file with phrases



(b) Input area of xterm with phrase input

Figure 9. Hanzi phrases in the input table and in xterm input area

REFERENCES

1. J. D. Becker, 'Typing Chinese, Japanese, and Korean', *IEEE Computer*, **18**, (1), 27-34 (1985).
2. N. P. Archer, M. W. L. Chan, S. J. Huang and R. T. Liu, 'A Chinese-English microcomputer system', *Comm. ACM*, **31**, (8), 977-982 (1988).
3. M. Tayli and A. I. Al-Salamah, 'Building bilingual microcomputer systems', *Comm. ACM*, **33**, (5), 495-504 (1990).
4. *X Protocol Manual*, Volume 0 of the X Window System Series, O'Reilly and Associates, Sebastopol, CA, 1990.
5. *Code of Chinese Graphic Character for Information Interchange, Primary Set (GB2312-80)*, National Standards Bureau, Beijing, China, 1980.
6. *Standard Code of Common Chinese Character for Information Interchange, CNS 11643*, Taiwan, 1986.
7. Ken R. Lunde, 'Electronic handling of Japanese text', Adobe Systems Incorporated, Mountain View, CA, 1992. (Electronic version available from Internet ftp site ucdavis.edu: pub/JIS/japan1.inf.)
8. J. J. T. Huang and T. D. Huang, *An Introduction to Chinese, Japanese, and Korean Computing*, World Scientific Computing, Singapore, 1989.
9. *The Status and Progress of Chinese Language Processing Technology*, Association for Common Chinese Code International, Beijing, China, March 1991.
10. R. Walters, 'Design of a bitmapped multilingual workstation', *IEEE Computer*, **23**, (2), 33-41 (1990).
11. M. Ishisone, *X11 Input Conversion*, Software Research Associates, Inc. Tokyo, Japan, 1989.
12. A. Nye, V. Quercia and T. O'Reilly, *XLib System Programming Manual, XLib Reference Manual, and X Window System User's Guide*, Volume 1, 2, 3 of the X Window System Series, O'Reilly and Associates, Sebastopol, CA, 1990.
13. G. A. Miller, 'The magic number seven, plus or minus two: some limits on our capacity for processing information', *Psychological Review*, **63**, 81-97 (1956).

14. A. V. Aho, J. E. Hopcroft and J. D. Ullman, *Data structures and Algorithms*, Addison-Wesley, MA, 1987.
15. Multilingual Computing Group, Institute of System Science, National University of Singapore, 'A multilingual computing environment based on X window system', *Computer Processing of Chinese and Oriental Languages*, 5, (1), 53-73 (1990).
16. kterm, contributed software in X11 Release 1 to Release 5, MIT, originally developed by the Japan Unix Society, Japan.
17. KABA software research group, 'Overview of GMW-Winn system', Research Institute for Mathematical Sciences, Kyoto University, Kyoto, Japan.
18. V. Joloboff and B. McMahon, 'X Window System, Version 11, input method specifications', *Public Review Draft*, X Consortium, MIT, Nov. 1990.
19. R. Stallman, *GNU emacs Manual*, Free Software Foundations, Cambridge, MA, 1986.

Exceptional C or C with Exceptions

N. H. GEHANI

AT&T Bell Laboratories, Rm 30-414, 600 Mountain Ave., Murray Hill, New Jersey
07974-2070, U.S.A.

SUMMARY

C does not have exception handling facilities. Errors are handled by examining the value returned by each function and signals (conditions reported to the program) are handled by using library functions. These approaches lead to *ad hoc* error-handling techniques and can make programs hard to understand. Exceptional C, a superset of C, provides exception handling facilities. Exceptional C integrates the two techniques used by C programmers (i.e., status values and signals) to handle errors into one unified exception handling mechanism. In this paper, I review exception handling models, specify the criteria used for designing the exception handling facilities in Exceptional C, and then describe these facilities. I also illustrate the use of the exception handling facilities with examples.

KEY WORDS Exception handling Error handling Fault tolerance

INTRODUCTION

As programs grow in size, programming complexity is significantly increased by the need to handle special cases and unusual or error conditions which crop up with startling frequency.¹ In fact, up to 50-70 per cent of the code in a large system can be code for handling errors. Error-handling code is a weak link in the development of robust programs: errors are not well understood and error-handling code is not well tested.² Consequently, as programs grow larger, a standard mechanism for handling exceptions becomes important.³

An *exception* is an error or an event that occurs unexpectedly or infrequently, such as division by zero, out-of-memory, or the premature interruption of program execution. Exception handling covers both the response to the exception and recovery from the error or the infrequent event. An *exception mechanism* is a language control structure that specifies that the standard continuation of a program be replaced by an exceptional continuation upon the detection of an exception.⁴ Exception-handling enhances program reliability and fault tolerance.⁵

Several languages (such as PL/I, Ada, Mesa, and CLU) provide explicit exception handling facilities but many languages (such as Pascal, C and FORTRAN) do not. In languages without exception handling facilities, other methods must be used to indicate exceptions and to handle exceptions. The most common method is to have functions return 'status values' or 'funny values'. If the value returned by a function indicates that an exception has occurred during the function call, then appropriate action is taken. The status-value technique^{5,6}

